C 41

41.1 CALL—Call Procedure

Opcode	Instruction	Description
E8 *cw*	CALL *rel16*	Call near, relative, displacement relative to next instruction
E8 *cd*	CALL *rel32*	Call near, relative, displacement relative to next instruction
FF /2	CALL *r/m16*	Call near, absolute indirect, address given in *r/m16*
FF /2	CALL *r/m32*	Call near, absolute indirect, address given in *r/m32*
9A *cd*	CALL *ptr16:16*	Call far, absolute, address given in operand
9A *cp*	CALL *ptr16:32*	Call far, absolute, address given in operand
FF /3	CALL *m16:16*	Call far, absolute indirect, address given in *m16:16*
FF /3	CALL *m16:32*	Call far, absolute indirect, address given in *m16:32*

Description

Saves procedure linking information on the stack and branches to the procedure (called procedure) specified with the destination (target) operand. The target operand specifies the address of the first instruction in the called procedure. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of calls:

- Near call—A call to a procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment call.
- Far call—A call to a procedure located in a different segment than the current code segment, sometimes referred to as an intersegment call.
- Inter-privilege-level far call—A far call to a procedure in a segment at a different privilege level than that of the currently executing program or procedure.
- Task switch—A call to a procedure located in a different task.

The latter two call types (inter-privilege-level call and task switch) can only be executed in protected mode. See the section titled “Calling Procedures Using CALL and RET”, for additional information on near, far, and inter-privilege-level calls. See Chapter 6, *Task Management*, in the *Intel Architecture Software Developer’s Manual, Volume 3*, for information on performing task switches with the CALL instruction.

Near Call. When executing a near call, the processor pushes the value of the EIP register (which contains the offset of the instruction following the CALL instruction) onto the stack (for use later as a return-instruction pointer). The processor then branches to the address in the current code segment specified with the target operand. The target operand specifies either an absolute offset in the code segment (that is an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register, which points to the instruction following the CALL instruction). The CS register is not changed on near calls.

For a near call, an absolute offset is specified indirectly in a general-purpose register or a memory location (*r/m16* or *r/m32*). The operand-size attribute determines the size of the target operand (16 or 32 bits). Absolute offsets are loaded directly into the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s, resulting in a maximum instruction pointer size of 16 bits. (When accessing an absolute offset indirectly using the stack pointer [ESP] as a base register, the base value used is the value of the ESP before the instruction executes.)

A relative offset (*rel16* or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 16- or 32-bit immediate value. This value is added to the value in the EIP register. As with absolute offsets, the operand-size attribute determines the size of the target operand (16 or 32 bits).

Far Calls in Real-Address or Virtual-8086 Mode. When executing a far call in real-address or virtual-8086 mode, the processor pushes the current value of both the CS and EIP registers onto the stack for use as a return-instruction pointer. The processor then performs a “far branch” to the code segment and offset specified with the target operand for the called procedure. Here the target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and offset of the called procedure is encoded in the instruction, using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s.

Far Calls in Protected Mode. When the processor is operating in protected mode, the CALL instruction can be used to perform the following three types of far calls:

- Far call to the same privilege level.
- Far call to a different privilege level (inter-privilege level call).
- Task switch (far call to another task).

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register, and the offset from the instruction is loaded into the EIP register.

Note that a call gate (described in the next paragraph) can also be used to perform far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making calls between 16-bit and 32-bit code segments.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a call gate. The segment selector specified by the target operand identifies the call gate. Here again, the target operand can specify the call gate segment selector either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The processor obtains the segment selector for the new code segment and the new

instruction pointer (offset) from the call gate descriptor. (The offset from the target operand is ignored when a call gate is used.) On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, no stack switch occurs.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack, an (optional) set of parameters from the calling procedures stack, and the segment selector and instruction pointer for the calling procedure's code segment. (A value in the call gate descriptor determines how many parameters to copy to the new stack.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Executing a task switch with the CALL instruction, is somewhat similar to executing a call through a call gate. Here the target operand specifies the segment selector of the task gate for the task being switched to (and the offset in the target operand is ignored.) The task gate in turn points to the TSS for the task, which contains the segment selectors for the task's code and stack segments. The TSS also contains the EIP value for the next instruction that was to be executed before the task was suspended. This instruction pointer value is loaded into EIP register so that the task begins executing again at this next instruction.

The CALL instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 6, *Task Management*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for detailed information on the mechanics of a task switch.

Note that when you execute at task switch with a CALL instruction, the nested task flag (NT) is set in the EFLAGS register and the new TSS's previous task link field is loaded with the old tasks TSS selector. Code is expected to suspend this nested task by executing an IRET instruction, which, because the NT flag is set, will automatically use the previous task link to return to the calling task. (See "Task Linking" in Chapter 6 of the *Intel Architecture Software Developer's Manual, Volume 3*, for more information on nested tasks.) Switching tasks with the CALL instruction differs in this regard from the JMP instruction which does not set the NT flag and therefore does not expect an IRET instruction to suspend the task.

Mixing 16-Bit and 32-Bit Calls. When making far calls between 16-bit and 32-bit code segments, the calls should be made through a call gate. If the far call is from a 32-bit code segment to a 16-bit code segment, the call should be made from the first 64 KBytes of the 32-bit code segment. This is because the operand-size attribute of the instruction is set to 16, so only a 16-bit return address offset is saved. Also, the call should be made using a 16-bit call gate so that 16-bit values will be pushed on the stack. See Chapter 16, *Mixing 16-Bit and 32-Bit Code*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for more information on making calls between 16-bit and 32-bit code segments.

Operation

```
IF near call
    THEN IF near relative call
         IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
         THEN IF OperandSize = 32
              THEN
                   IF stack not large enough for a 4-byte return address THEN #SS(0); FI;
                   Push(EIP);
                   EIP ← EIP + DEST; (* DEST is rel32 *)
              ELSE (* OperandSize = 16 *)
                   IF stack not large enough for a 2-byte return address THEN #SS(0); FI;
                   Push(IP);
                   EIP ← (EIP + DEST) AND 0000FFFFH; (* DEST is rel16 *)
         FI;
    FI;
    ELSE (* near absolute call *)
         IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
         IF OperandSize = 32
```

```
                THEN
                    IF stack not large enough for a 4-byte return address THEN #SS(0); FI;
                    Push(EIP);
                    EIP ← DEST; (* DEST is r/m32 *)
                ELSE (* OperandSize = 16 *)
                    IF stack not large enough for a 2-byte return address THEN #SS(0); FI;
                    Push(IP);
                    EIP ← DEST AND 0000FFFFH; (* DEST is r/m16 *)
            FI;
        FI:
FI;

IF far call AND (PE = 0 OR (PE = 1 AND VM = 1)) (* real-address or virtual-8086 mode *)
        THEN
            IF OperandSize = 32
                THEN
                    IF stack not large enough for a 6-byte return address THEN #SS(0); FI;
                    IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
                    Push(CS); (* padded with 16 high-order bits *)
                    Push(EIP);
                    CS ← DEST[47:32]; (* DEST is ptr16:32 or [m16:32] *)
                    EIP ← DEST[31:0]; (* DEST is ptr16:32 or [m16:32] *)
                ELSE (* OperandSize = 16 *)
                    IF stack not large enough for a 4-byte return address THEN #SS(0); FI;
                    IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
                    Push(CS);
                    Push(IP);
                    CS ← DEST[31:16]; (* DEST is ptr16:16 or [m16:16] *)
                    EIP ← DEST[15:0]; (* DEST is ptr16:16 or [m16:16] *)
                    EIP ← EIP AND 0000FFFFH; (* clear upper 16 bits *)
            FI;
FI;

IF far call AND (PE = 1 AND VM = 0) (* Protected mode, not virtual-8086 mode *)
        THEN
            IF segment selector in target operand null THEN #GP(0); FI;
            IF segment selector index not within descriptor table limits
                THEN #GP(new code segment selector);
            FI;
            Read type and access rights of selected segment descriptor;
            IF segment type is not a conforming or nonconforming code segment, call gate,
                task gate, or TSS THEN #GP(segment selector); FI;
            Depending on type and access rights
                GO TO CONFORMING-CODE-SEGMENT;
                GO TO NONCONFORMING-CODE-SEGMENT;
                GO TO CALL-GATE;
                GO TO TASK-GATE;
                GO TO TASK-STATE-SEGMENT;
FI;

CONFORMING-CODE-SEGMENT:
    IF DPL > CPL THEN #GP(new code segment selector); FI;
    IF segment not present THEN #NP(new code segment selector); FI;
    IF OperandSize = 32
        THEN
            IF stack not large enough for a 6-byte return address THEN #SS(0); FI;
            IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
            Push(CS); (* padded with 16 high-order bits *)
            Push(EIP);
            CS ← DEST(NewCodeSegmentSelector);
            (* segment descriptor information also loaded *)
            CS(RPL) ← CPL
            EIP ← DEST(offset);
        ELSE (* OperandSize = 16 *)
            IF stack not large enough for a 4-byte return address THEN #SS(0); FI;
            IF the instruction pointer is not within code segment limit THEN #GP(0); FI;
            Push(CS);
            Push(IP);
            CS ← DEST(NewCodeSegmentSelector);
            (* segment descriptor information also loaded *)
            CS(RPL) ← CPL
            EIP ← DEST(offset) AND 0000FFFFH; (* clear upper 16 bits *)
    FI;
END;

NONCONFORMING-CODE-SEGMENT:
    IF (RPL > CPL) OR (DPL ≠ CPL) THEN #GP(new code segment selector); FI;
    IF segment not present THEN #NP(new code segment selector); FI;
    IF stack not large enough for return address THEN #SS(0); FI;
    tempEIP ← DEST(offset)
    IF OperandSize=16
        THEN
            tempEIP ← tempEIP AND 0000FFFFH; (* clear upper 16 bits *)
    FI;
    IF tempEIP outside code segment limit THEN #GP(0); FI;
    IF OperandSize = 32
```

```
            THEN
                Push(CS); (* padded with 16 high-order bits *)
                Push(EIP);
                CS ← DEST(NewCodeSegmentSelector);
                (* segment descriptor information also loaded *)
                CS(RPL) ← CPL;
                EIP ← tempEIP;
            ELSE (* OperandSize = 16 *)
                Push(CS);
                Push(IP);
                CS ← DEST(NewCodeSegmentSelector);
                (* segment descriptor information also loaded *)
                CS(RPL) ← CPL;
                EIP ← tempEIP;
        FI;
END;

CALL-GATE:
        IF call gate DPL < CPL or RPL THEN #GP(call gate selector); FI;
        IF call gate not present THEN #NP(call gate selector); FI;
        IF call gate code-segment selector is null THEN #GP(0); FI;
        IF call gate code-segment selector index is outside descriptor table limits
            THEN #GP(code segment selector); FI;
        Read code segment descriptor;
        IF code-segment segment descriptor does not indicate a code segment
        OR code-segment segment descriptor DPL > CPL
            THEN #GP(code segment selector); FI;
        IF code segment not present THEN #NP(new code segment selector); FI;
        IF code segment is non-conforming AND DPL < CPL
            THEN go to MORE-PRIVILEGE;
            ELSE go to SAME-PRIVILEGE;
        FI;
END;

MORE-PRIVILEGE:
        IF current TSS is 32-bit TSS
            THEN
                TSSstackAddress ← new code segment (DPL * 8) + 4
                IF (TSSstackAddress + 7) > TSS limit
                    THEN #TS(current TSS selector); FI;
                newSS ← TSSstackAddress + 4;
                newESP ← stack address;
            ELSE (* TSS is 16-bit *)
                TSSstackAddress ← new code segment (DPL * 4) + 2
                IF (TSSstackAddress + 4) > TSS limit
                    THEN #TS(current TSS selector); FI;
                newESP ← TSSstackAddress;
                newSS ← TSSstackAddress + 2;
        FI;
        IF stack segment selector is null THEN #TS(stack segment selector); FI;
        IF stack segment selector index is not within its descriptor table limits
            THEN #TS(SS selector); FI
        Read code segment descriptor;
        IF stack segment selector's RPL ≠ DPL of code segment
            OR stack segment DPL ≠ DPL of code segment
            OR stack segment is not a writable data segment
                THEN #TS(SS selector); FI
        IF stack segment not present THEN #SS(SS selector); FI;
        IF CallGateSize = 32
            THEN
                IF stack does not have room for parameters plus 16 bytes
                    THEN #SS(SS selector); FI;
                IF CallGate(InstructionPointer) not within code segment limit THEN #GP(0); FI;
                SS ← newSS;
                (* segment descriptor information also loaded *)
                ESP ← newESP;
                CS:EIP ← CallGate(CS:InstructionPointer);
                (* segment descriptor information also loaded *)
                Push(oldSS:oldESP); (* from calling procedure *)
                temp ← parameter count from call gate, masked to 5 bits;
                Push(parameters from calling procedure's stack, temp)
                Push(oldCS:oldEIP); (* return address to calling procedure *)
            ELSE (* CallGateSize = 16 *)
                IF stack does not have room for parameters plus 8 bytes
                    THEN #SS(SS selector); FI;
                IF (CallGate(InstructionPointer) AND FFFFH) not within code segment limit
                    THEN #GP(0); FI;
                SS ← newSS;
                (* segment descriptor information also loaded *)
                ESP ← newESP;
                CS:IP ← CallGate(CS:InstructionPointer);
                (* segment descriptor information also loaded *)
                Push(oldSS:oldESP); (* from calling procedure *)
                temp ← parameter count from call gate, masked to 5 bits;
                Push(parameters from calling procedure's stack, temp)
                Push(oldCS:oldEIP); (* return address to calling procedure *)
```

```
    FI;
    CPL ← CodeSegment(DPL)

    IF task gate not present
        THEN #NP(task gate selector);
    FI;
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
        OR index not within GDT limits
            THEN #GP(TSS selector);
    FI;
    Access TSS descriptor in GDT;

    IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
            THEN #GP(TSS selector);
    FI;
    IF TSS not present
        THEN #NP(TSS selector);
    FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0);
    FI;
END;

TASK-STATE-SEGMENT:
    IF TSS DPL < CPL or RPL
    OR TSS descriptor indicates TSS not available
        THEN #GP(TSS selector);
    FI;
    IF TSS is not present
        THEN #NP(TSS selector);
    FI;
    SWITCH-TASKS (with nesting) to TSS
    IF EIP not within code segment limit
        THEN #GP(0);
    FI;
END;
```

Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

Protected Mode Exceptions

#GP(0)	If target offset in destination operand is beyond the new code segment limit.
	If the segment selector in the destination operand is null.
	If the code segment selector in the gate is null.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#GP(selector)	If code segment or gate or TSS selector index is outside descriptor table limits.
	If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.
	If the DPL for a nonconforming-code segment is not equal to the CPL or the RPL for the segment's segment selector is greater than the CPL.
	If the DPL for a conforming-code segment is greater than the CPL.
	If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector.

	If the segment descriptor for a segment selector from a call gate does not indicate it is a code segment.
	If the segment selector from a call gate is beyond the descriptor table limits.
	If the DPL for a code-segment obtained from a call gate is greater than the CPL.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is busy or not available.
#SS(0)	If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when no stack switch occurs.
	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when a stack switch occurs.
	If the SS register is being loaded as part of a stack switch and the segment pointed to is marked not present.
	If stack segment does not have room for the return address, parameters, or stack segment pointer, when stack switch occurs.
#NP(selector)	If a code segment, data segment, stack segment, call gate, task gate, or TSS is not present.
#TS(selector)	If the new stack segment selector and ESP are beyond the end of the TSS.
	If the new stack segment selector is null.
	If the RPL of the new stack segment selector in the TSS is not equal to the DPL of the code segment being accessed.
	If DPL of the stack segment descriptor for the new stack segment is not equal to the DPL of the code segment descriptor.
	If the new stack segment is not a writable data segment.
	If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when the CPL is 3 and alignment checking is enabled.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the target offset is beyond the code segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the target offset is beyond the code segment limit.
#PF(fault-code)	If a page fault occurs.

#AC(0) If an unaligned memory access occurs when alignment checking is enabled.

41.2 CBW/CWDE—Convert Byte to Word/Convert Word to Doubleword

Opcode	Instruction	Description
98	CBW	AX ← sign-extend of AL
98	CWDE	EAX ← sign-extend of AX

Description

Double the size of the source operand by means of sign extension (see “Sign Extension”). The CBW (convert byte to word) instruction copies the sign (bit 7) in the source operand into every bit in the AH register. The CWDE (convert word to doubleword) instruction copies the sign (bit 15) of the word in the AX register into the higher 16 bits of the EAX register.

The CBW and CWDE mnemonics reference the same opcode. The CBW instruction is intended for use when the operand-size attribute is 16 and the CWDE instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when CBW is used and to 32 when CWDE is used. Others may treat these mnemonics as synonyms (CBW/CWDE) and use the current setting of the operand-size attribute to determine the size of values to be converted, regardless of the mnemonic used.

The CWDE instruction is different from the CWD (convert word to double) instruction. The CWD instruction uses the DX:AX register pair as a destination operand; whereas, the CWDE instruction uses the EAX register as a destination.

Operation

```
IF OperandSize = 16 (* instruction = CBW *)
    THEN AX ← SignExtend(AL);
    ELSE (* OperandSize = 32, instruction = CWDE *)
        EAX ← SignExtend(AX);
FI;
```

Flags Affected

None.

Exceptions (All Operating Modes)

None.

41.3 CDQ—Convert Double to Quad

See entry for CWD/CDQ — Convert Word to Doubleword/Convert Doubleword to Quadword.

41.4 CLC—Clear Carry Flag

Opcode	Instruction	Description
F8	CLC	Clear CF flag

Description

Clears the CF flag in the EFLAGS register.

Operation

```
CF ← 0;
```

Flags Affected

The CF flag is cleared to 0. The OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

None.

41.5 CLD—Clear Direction Flag

Opcode	Instruction	Description
FC	CLD	Clear DF flag

Description

Clears the DF flag in the EFLAGS register. When the DF flag is set to 0, string operations increment the index registers (ESI and/or EDI).

Operation

```
DF ← 0;
```

Flags Affected

The DF flag is cleared to 0. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

None.

41.6 CLI—Clear Interrupt Flag

Opcode	Instruction	Description
FA	CLI	Clear interrupt flag; interrupts disabled when interrupt flag cleared

Description

Clears the IF flag in the EFLAGS register. No other flags are affected. Clearing the IF flag causes the processor to ignore maskable external interrupts. The IF flag and the CLI and STI instruction have no affect on the generation of exceptions and NMI interrupts.

The following decision table indicates the action of the CLI instruction (bottom of the table) depending on the processor's mode of operating and the CPL and IOPL of the currently running program or procedure (top of the table).

PE =	0	1	1	1	1
VM =	X	0	X	0	1
CPL	X	≤ IOPL	X	> IOPL	X
IOPL	X	X	= 3	X	< 3
IF ← 0	Y	Y	Y	N	N
#GP(0)	N	N	N	Y	Y

NOTES:
1. XDon't care
2. NAction in column 1 not taken
3. YAction in column 1 taken

Operation

```
IF PE = 0 (* Executing in real-address mode *)
     THEN
          IF ← 0;
     ELSE
          IF VM = 0   (* Executing in protected mode *)
               THEN
                    IF CPL ≤ IOPL
                         THEN
                              IF ← 0;
                         ELSE
                              #GP(0);
                    FI;
          FI;
               ELSE  (* Executing in Virtual-8086 mode  *)
                    IF IOPL = 3
                         THEN
                              IF  ←0
                         ELSE
                              #GP(0);
                    FI;
          FI;
FI;
```

Flags Affected

The IF is cleared to 0 if the CPL is equal to or less than the IOPL; otherwise, it is not affected. The other flags in the EFLAGS register are unaffected.

Protected Mode Exceptions

#GP(0) If the CPL is greater (has less privilege) than the IOPL of the current program or procedure.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

#GP(0) If the CPL is greater (has less privilege) than the IOPL of the current program or procedure.

41.7 CLTS—Clear Task-Switched Flag in CR0

Opcode	Instruction	Description
0F 06	CLTS	Clears TS flag in CR0

Description

Clears the task-switched (TS) flag in the CR0 register. This instruction is intended for use in operating-system procedures. It is a privileged instruction that can only be executed at a CPL of 0. It is allowed to be executed in real-address mode to allow initialization for protected mode.

The processor sets the TS flag every time a task switch occurs. The flag is used to synchronize the saving of FPU context in multitasking applications. See the description of the TS flag in the section titled “Control Registers” in Chapter 2 of the *Intel Architecture Software Developer’s Manual, Volume 3*, for more information about this flag.

Operation

```
CR0(TS) ← 0;
```

Flags Affected

The TS flag in CR0 register is cleared.

Protected Mode Exceptions

#GP(0) If the CPL is greater than 0.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

#GP(0) If the CPL is greater than 0.

41.8 CMC—Complement Carry Flag

Opcode	Instruction	Description
F5	CMC	Complement CF flag

Description

Complements the CF flag in the EFLAGS register.

Operation

```
CF ← NOT CF;
```

Flags Affected

The CF flag contains the complement of its original value. The OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

None.

41.9 CMOV*cc*—Conditional Move

Opcode	Instruction	Description
0F 47 */r*	CMOVA *r16, r/m16*	Move if above (CF=0 and ZF=0)
0F 47 */r*	CMOVA *r32, r/m32*	Move if above (CF=0 and ZF=0)
0F 43 */r*	CMOVAE *r16, r/m16*	Move if above or equal (CF=0)
0F 43 */r*	CMOVAE *r32, r/m32*	Move if above or equal (CF=0)
0F 42 */r*	CMOVB *r16, r/m16*	Move if below (CF=1)
0F 42 */r*	CMOVB *r32, r/m32*	Move if below (CF=1)
0F 46 */r*	CMOVBE *r16, r/m16*	Move if below or equal (CF=1 or ZF=1)
0F 46 */r*	CMOVBE *r32, r/m32*	Move if below or equal (CF=1 or ZF=1)
0F 42 */r*	CMOVC *r16, r/m16*	Move if carry (CF=1)
0F 42 */r*	CMOVC *r32, r/m32*	Move if carry (CF=1)
0F 44 */r*	CMOVE *r16, r/m16*	Move if equal (ZF=1)
0F 44 */r*	CMOVE *r32, r/m32*	Move if equal (ZF=1)
0F 4F */r*	CMOVG *r16, r/m16*	Move if greater (ZF=0 and SF=OF)
0F 4F */r*	CMOVG *r32, r/m32*	Move if greater (ZF=0 and SF=OF)
0F 4D */r*	CMOVGE *r16, r/m16*	Move if greater or equal (SF=OF)
0F 4D */r*	CMOVGE *r32, r/m32*	Move if greater or equal (SF=OF)
0F 4C */r*	CMOVL *r16, r/m16*	Move if less (SF<>OF)
0F 4C */r*	CMOVL *r32, r/m32*	Move if less (SF<>OF)
0F 4E */r*	CMOVLE *r16, r/m16*	Move if less or equal (ZF=1 or SF<>OF)
0F 4E */r*	CMOVLE *r32, r/m32*	Move if less or equal (ZF=1 or SF<>OF)
0F 46 */r*	CMOVNA *r16, r/m16*	Move if not above (CF=1 or ZF=1)
0F 46 */r*	CMOVNA *r32, r/m32*	Move if not above (CF=1 or ZF=1)
0F 42 */r*	CMOVNAE *r16, r/m16*	Move if not above or equal (CF=1)

0F 42 */r*	CMOVNAE *r32, r/m32*	Move if not above or equal (CF=1)
0F 43 */r*	CMOVNB *r16, r/m16*	Move if not below (CF=0)
0F 43 */r*	CMOVNB *r32, r/m32*	Move if not below (CF=0)
0F 47 */r*	CMOVNBE *r16, r/m16*	Move if not below or equal (CF=0 and ZF=0)
0F 47 */r*	CMOVNBE *r32, r/m32*	Move if not below or equal (CF=0 and ZF=0)
0F 43 */r*	CMOVNC *r16, r/m16*	Move if not carry (CF=0)
0F 43 */r*	CMOVNC *r32, r/m32*	Move if not carry (CF=0)
0F 45 */r*	CMOVNE *r16, r/m16*	Move if not equal (ZF=0)
0F 45 */r*	CMOVNE *r32, r/m32*	Move if not equal (ZF=0)
0F 4E */r*	CMOVNG *r16, r/m16*	Move if not greater (ZF=1 or SF<>OF)
0F 4E */r*	CMOVNG *r32, r/m32*	Move if not greater (ZF=1 or SF<>OF)
0F 4C */r*	CMOVNGE *r16, r/m16*	Move if not greater or equal (SF<>OF)
0F 4C */r*	CMOVNGE *r32, r/m32*	Move if not greater or equal (SF<>OF)
0F 4D */r*	CMOVNL *r16, r/m16*	Move if not less (SF=OF)
0F 4D */r*	CMOVNL *r32, r/m32*	Move if not less (SF=OF)
0F 4F */r*	CMOVNLE *r16, r/m16*	Move if not less or equal (ZF=0 and SF=OF)
0F 4F */r*	CMOVNLE *r32, r/m32*	Move if not less or equal (ZF=0 and SF=OF)

Opcode	Instruction	Description
0F 41 */r*	CMOVNO *r16, r/m16*	Move if not overflow (OF=0)
0F 41 */r*	CMOVNO *r32, r/m32*	Move if not overflow (OF=0)
0F 4B */r*	CMOVNP *r16, r/m16*	Move if not parity (PF=0)
0F 4B */r*	CMOVNP *r32, r/m32*	Move if not parity (PF=0)
0F 49 */r*	CMOVNS *r16, r/m16*	Move if not sign (SF=0)
0F 49 */r*	CMOVNS *r32, r/m32*	Move if not sign (SF=0)
0F 45 */r*	CMOVNZ *r16, r/m16*	Move if not zero (ZF=0)
0F 45 */r*	CMOVNZ *r32, r/m32*	Move if not zero (ZF=0)
0F 40 */r*	CMOVO *r16, r/m16*	Move if overflow (OF=0)
0F 40 */r*	CMOVO *r32, r/m32*	Move if overflow (OF=0)
0F 4A */r*	CMOVP *r16, r/m16*	Move if parity (PF=1)
0F 4A */r*	CMOVP *r32, r/m32*	Move if parity (PF=1)
0F 4A */r*	CMOVPE *r16, r/m16*	Move if parity even (PF=1)
0F 4A */r*	CMOVPE *r32, r/m32*	Move if parity even (PF=1)
0F 4B */r*	CMOVPO *r16, r/m16*	Move if parity odd (PF=0)
0F 4B */r*	CMOVPO *r32, r/m32*	Move if parity odd (PF=0)
0F 48 */r*	CMOVS *r16, r/m16*	Move if sign (SF=1)
0F 48 */r*	CMOVS *r32, r/m32*	Move if sign (SF=1)
0F 44 */r*	CMOVZ *r16, r/m16*	Move if zero (ZF=1)
0F 44 */r*	CMOVZ *r32, r/m32*	Move if zero (ZF=1)

Description

The CMOV*cc* instructions check the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and perform a move operation if the flags are in a specified state (or condition). A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, a move is not performed and execution continues with the instruction following the CMOV*cc* instruction.

These instructions can move a 16- or 32-bit value from memory to a general-purpose register or from one general-purpose register to another. Conditional moves of 8-bit register operands are not supported.

The conditions for each CMOV*cc* mnemonic is given in the description column of the above table. The terms “less” and “greater” are used for comparisons of signed integers and the terms “above” and “below” are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the CMOVA (conditional move if above) instruction and the CMOVNBE (conditional move if not below or equal) instruction are alternate mnemonics for the opcode 0F 47H.

The CMOV*cc* instructions are new for the Pentium Pro processor family; however, they may not be supported by all the processors in the family. Software can determine if the CMOV*cc* instructions are supported by checking the processor’s feature information with the CPUID instruction (see “CPUID—CPU Identification” in this chapter).

Operation

```
temp ← DEST
IF condition TRUE
    THEN
        DEST ← SRC
    ELSE
        DEST ← temp
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.

#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

41.10 CMP—Compare Two Operands

Opcode	Instruction	Description
3C *ib*	CMP AL, *imm8*	Compare *imm8* with AL
3D *iw*	CMP AX, *imm16*	Compare *imm16* with AX
3D *id*	CMP EAX, *imm32*	Compare *imm32* with EAX
80 /7 *ib*	CMP *r/m8, imm8*	Compare *imm8* with *r/m8*
81 /7 *iw*	CMP *r/m16, imm16*	Compare *imm16* with *r/m16*
81 /7 *id*	CMP *r/m32,imm32*	Compare *imm32* with *r/m32*
83 /7 *ib*	CMP *r/m16,imm8*	Compare *imm8* with *r/m16*
83 /7 *ib*	CMP *r/m32,imm8*	Compare *imm8* with *r/m32*
38 /*r*	CMP *r/m8,r8*	Compare *r8* with *r/m8*
39 /*r*	CMP *r/m16,r16*	Compare *r16* with *r/m16*
39 /*r*	CMP *r/m32,r32*	Compare *r32* with *r/m32*
3A /*r*	CMP *r8,r/m8*	Compare *r/m8* with *r8*
3B /*r*	CMP *r16,r/m16*	Compare *r/m16* with *r16*
3B /*r*	CMP *r32,r/m32*	Compare *r/m32* with *r32*

Description

Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand.

The CMP instruction is typically used in conjunction with a conditional jump (J*cc*), condition move (CMOV*cc*), or SET*cc* instruction. The condition codes used by the J*cc*, CMOV*cc*, and SET*cc* instructions are based on the results of a CMP instruction. “EFLAGS Cross-Reference and Condition Codes” shows the relationship of the status flags and the condition codes.

Operation

```
temp ← SRC1 – SignExtend(SRC2);
ModifyStatusFlags; (* Modify status flags in the same manner as the SUB instruction*)
```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.

#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

41.11 CMPS/CMPSB/CMPSW/CMPSD—Compare String Operands

Opcode	Instruction	Description
A6	CMPS m8, m8	Compares byte at address DS:(E)SI with byte at address ES:(E)DI and sets the status flags accordingly
A7	CMPS m16, m16	Compares word at address DS:(E)SI with word at address ES:(E)DI and sets the status flags accordingly
A7	CMPS m32, m32	Compares doubleword at address DS:(E)SI with doubleword at address ES:(E)DI and sets the status flags accordingly
A6	CMPSB	Compares byte at address DS:(E)SI with byte at address ES:(E)DI and sets the status flags accordingly
A7	CMPSW	Compares word at address DS:(E)SI with word at address ES:(E)DI and sets the status flags accordingly
A7	CMPSD	Compares doubleword at address DS:(E)SI with doubleword at address ES:(E)DI and sets the status flags accordingly

Description

Compares the byte, word, or double word specified with the first source operand with the byte, word, or double word specified with the second source operand and sets the status flags in the EFLAGS register according to the results. Both the source operands are located in memory. The address of the first source operand is read from either the DS:ESI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The address of the second source operand is read from either the ES:EDI or the ES:DI registers (again depending on the address-size attribute of the instruction). The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the CMPS mnemonic) allows the two source operands to be specified explicitly. Here, the source operands should be symbols that indicate the size and location of the source values. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbols must specify the correct **type** (size) of the operands (bytes, words, or doublewords), but they do not have to specify the correct **location**. The locations of the source operands are always specified by the DS:(E)SI and ES:(E)DI registers, which must be loaded correctly before the compare string instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the CMPS instructions. Here also the DS:(E)SI and ES:(E)DI registers are assumed by the processor to specify the location of the source operands. The size of the source operands is selected with the mnemonic: CMPSB (byte comparison), CMPSW (word comparison), or CMPSD (doubleword comparison).

After the comparison, the (E)SI and (E)DI registers are incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI and (E)DI register are incremented; if the DF flag is 1, the (E)SI and (E)DI registers are decremented.) The registers are incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The CMPS, CMPSB, CMPSW, and CMPSD instructions can be preceded by the REP prefix for block comparisons of ECX bytes, words, or doublewords. More often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of the status flags before the next comparison is made. See “REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix” in this chapter for a description of the REP prefix.

Operation

```
temp ←SRC1 − SRC2;
SetStatusFlags(temp);
IF (byte comparison)
    THEN IF DF = 0
        THEN
            (E)SI ← (E)SI + 1;
            (E)DI ← (E)DI + 1;
        ELSE
            (E)SI ← (E)SI − 1;
            (E)DI ← (E)DI − 1;
        FI;
    ELSE IF (word comparison)
        THEN IF DF = 0
            (E)SI ← (E)SI + 2;
            (E)DI ← (E)DI + 2;
        ELSE
            (E)SI ← (E)SI − 2;
            (E)DI ← (E)DI − 2;
        FI;
    ELSE (* doubleword comparison*)
        THEN IF DF = 0
            (E)SI ← (E)SI + 4;
            (E)DI ← (E)DI + 4;
        ELSE
            (E)SI ← (E)SI − 4;
            (E)DI ← (E)DI − 4;
        FI;
FI;
```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are set according to the temporary result of the comparison.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

41.12 CMPXCHG—Compare and Exchange

Opcode	Instruction	Description
0F B0/*r*	CMPXCHG *r/m8,r8*	Compare AL with *r/m8*. If equal, ZF is set and *r8* is loaded into *r/m8*. Else, clear ZF and load *r/m8* into AL.
0F B1/*r*	CMPXCHG *r/m16,r16*	Compare AX with *r/m16*. If equal, ZF is set and *r16* is loaded into *r/m16*. Else, clear ZF and load *r/m16* into AL
0F B1/*r*	CMPXCHG *r/m32,r32*	Compare EAX with *r/m32*. If equal, ZF is set and *r32* is loaded into *r/m32*. Else, clear ZF and load *r/m32* into AL

Description

Compares the value in the AL, AX, or EAX register (depending on the size of the operand) with the first operand (destination operand). If the two values are equal, the second operand (source operand) is loaded into the destination operand. Otherwise, the destination operand is loaded into the AL, AX, or EAX register.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

Intel Architecture Compatibility

This instruction is not supported on Intel processors earlier than the Intel486 processors.

Operation

```
(* accumulator = AL, AX, or EAX, depending on whether *)
(* a byte, word, or doubleword comparison is being performed*)
IF accumulator = DEST
    THEN
        ZF ← 1
        DEST ← SRC
    ELSE
        ZF ← 0
        accumulator ← DEST
FI;
```

Flags Affected

The ZF flag is set if the values in the destination operand and register AL, AX, or EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are set according to the results of the comparison operation.

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

41.13 CMPXCHG8B—Compare and Exchange 8 Bytes

Opcode	Instruction	Description
0F C7 /1 m64	CMPXCHG8B *m64*	Compare EDX:EAX with *m64*. If equal, set ZF and load ECX:EBX into *m64*. Else, clear ZF and load *m64* into EDX:EAX.

Description

Compares the 64-bit value in EDX:EAX with the operand (destination operand). If the values are equal, the 64-bit value in ECX:EBX is stored in the destination operand. Otherwise, the value in the destination operand is loaded into EDX:EAX. The destination operand is an 8-byte memory location. For the EDX:EAX and ECX:EBX register pairs, EDX and ECX contain the high-order 32 bits and EAX and EBX contain the low-order 32 bits of a 64-bit value.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

Intel Architecture Compatibility

This instruction is not supported on Intel processors earlier than the Pentium processors.

Operation

```
IF (EDX:EAX = DEST)
    ZF ← 1
    DEST ← ECX:EBX
ELSE
    ZF ← 0
    EDX:EAX ← DEST
```

Flags Affected

The ZF flag is set if the destination operand and EDX:EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are unaffected.

Protected Mode Exceptions

#UD	If the destination operand is not a memory location.
#GP(0)	If the destination is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#UD	If the destination operand is not a memory location.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#UD	If the destination operand is not a memory location.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

41.14 CPUID—CPU Identification

Opcode	Instruction	Description
0F A2	CPUID	EAX ← Processor identification information

Description

Provides processor identification information in registers EAX, EBX, ECX, and EDX. This information identifies Intel as the vendor, gives the family, model, and stepping of processor, feature information, and cache information. An input value loaded into the EAX register determines what information is returned, as shown in Table 41-1.

Table 41-1. Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
0	EAX	Maximum CPUID Input Value (2 for the Pentium® Pro processor and 1 for the Pentium processor and the later versions of Intel486™ processor that support the CPUID instruction).
	EBX	“Genu”
	ECX	“ntel”
	EDX	“inel”
1	EAX	Version Information (Type, Family, Model, and Stepping ID)
	EBX	Reserved
	ECX	Reserved
	EDX	Feature Information
2	EAX	Cache and TLB Information
	EBX	Cache and TLB Information
	ECX	Cache and TLB Information
	EDX	Cache and TLB Information

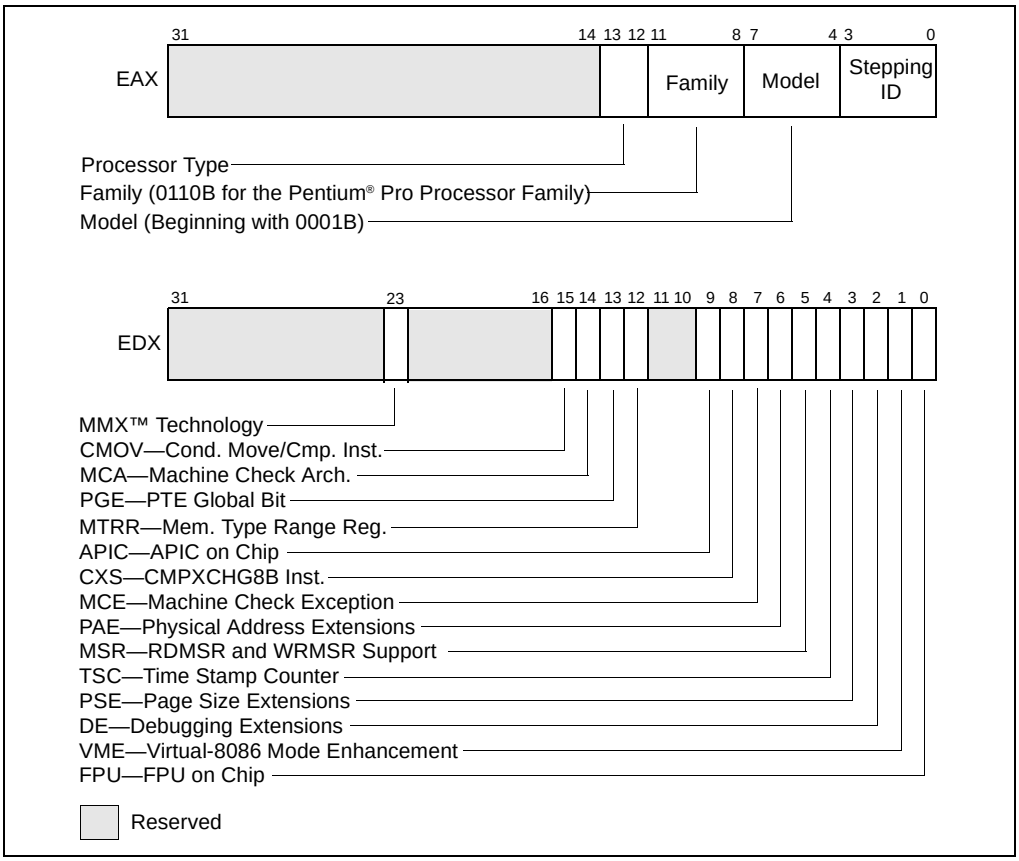
The CPUID instruction can be executed at any privilege level to serialize instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed (see "Serializing Instructions" in Chapter 7 of the *Intel Architecture Software Developer's Manual, Volume 3*).

When the input value in register EAX is 0, the processor returns the highest value the CPUID instruction recognizes in the EAX register (see Table 41-1). A vendor identification string is returned in the EBX, EDX, and ECX registers. For Intel processors, the vendor identification string is "GenuineIntel" as follows:

```
EBX ← 756e6547h (* "Genu", with G in the low nibble of BL *)
EDX ← 49656e69h (* "ineI", with i in the low nibble of DL *)
ECX ← 6c65746eh (* "ntel", with n in the low nibble of CL *)
```

When the input value is 1, the processor returns version information in the EAX register and feature information in the EDX register (see Figure 41-1).

Figure 41-1. Version and Feature Information in Registers EAX and EDX



The version information consists of an Intel Architecture family identifier, a model identifier, a stepping ID, and a processor type. The model, family, and processor type for the first processor in the Intel Pentium Pro family is as follows:

- Model—0001B
- Family—0110B
- Processor Type—00B

See AP-485, *Intel Processor Identification and the CPUID Instruction* (Order Number 241618), the *Intel Pentium® Pro Processor Specification Update* (Order Number 242689), and the *Intel Pentium® Processor Specification Update* (Order Number 242480) for more information on identifying earlier Intel Architecture processors.

The available processor types are given in Table 41-2. Intel releases information on stepping IDs as needed.

Table 41-2. Processor Type Field

Type	Encoding
Original OEM Processor	00B
Intel OverDrive® Processor	01B
Dual processor*	10B
Intel reserved.	11B

NOTE: * Not applicable to Intel386™ and Intel486™ processors.

Table 41-3 shows the encoding of the feature flags in the EDX register. A feature flag set to 1 indicates the corresponding feature is supported. Software should identify Intel as the vendor to properly interpret the feature flags.

Table 41-3. Feature Flags Returned in EDX Register (Sheet 1 of 2)

Bit	Feature	Description
0	FPU—Floating-Point Unit on Chip	Processor contains an FPU and executes the Intel 387 instruction set.
1	VME—Virtual-8086 Mode Enhancements	Processor supports the following virtual-8086 mode enhancements: • CR4.VME bit enables virtual-8086 mode extensions. • CR4.PVI bit enables protected-mode virtual interrupts. • Expansion of the TSS with the software indirection bitmap. • EFLAGS.VIF bit (virtual interrupt flag). • EFLAGS.VIP bit (virtual interrupt pending flag).
2	DE—Debugging Extensions	Processor supports I/O breakpoints, including the CR4.DE bit for enabling debug extensions and optional trapping of access to the DR4 and DR5 registers.
3	PSE—Page Size Extensions	Processor supports 4-Mbyte pages, including the CR4.PSE bit for enabling page size extensions, the modified bit in page directory entries (PDEs), page directory entries, and page table entries (PTEs).
4	TSC—Time Stamp Counter	Processor supports the RDTSC (read time stamp counter) instruction, including the CR4.TSD bit that, along with the CPL, controls whether the time stamp counter can be read.
5	MSR—Model Specific Registers	Processor supports the RDMSR (read model-specific register) and WRMSR (write model-specific register) instructions.
6	PAE—Physical Address Extension	Processor supports physical addresses greater than 32 bits, the extended page-table-entry format, an extra level in the page translation tables, and 2-MByte pages. The CR4.PAE bit enables this feature. The number of address bits is implementation specific. The Pentium® Pro processor supports 36 bits of addressing when the PAE bit is set.
7	MCE—Machine Check Exception	Processor supports the CR4.MCE bit, enabling machine check exceptions. However, this feature does not define the model-specific implementations of machine-check error logging, reporting, or processor shutdowns. Machine-check exception handlers might have to check the processor version to do model-specific processing of the exception or check for the presence of the standard machine-check feature.

Table 41-3. Feature Flags Returned in EDX Register (Sheet 2 of 2)

Bit	Feature	Description
8	CX8—CMPXCHG8B Instruction	Processor supports the CMPXCHG8B (compare and exchange 8 bytes) instruction.
9	APIC	Processor contains an on-chip Advanced Programmable Interrupt Controller (APIC) and it has been enabled and is available for use.
10,11	Reserved	
12	MTRR—Memory Type Range Registers	Processor supports machine-specific memory-type range registers (MTRRs). The MTRRs contains bit fields that indicate the processor's MTRR capabilities, including which memory types the processor supports, the number of variable MTRRs the processor supports, and whether the processor supports fixed MTRRs.
13	PGE—PTE Global Flag	Processor supports the CR4.PGE flag enabling the global bit in both PTDEs and PTEs. These bits are used to indicate translation lookaside buffer (TLB) entries that are common to different tasks and need not be flushed when control register CR3 is written.
14	MCA—Machine Check Architecture	Processor supports the MCG_CAP (machine check global capability) MSR. The MCG_CAP register indicates how many banks of error reporting MSRs the processor supports.
15	CMOV—Conditional Move and Compare Instructions	Processor supports the CMOV*cc* instruction and, if the FPU feature flag (bit 0) is also set, supports the FCMOV*cc* and FCOMI instructions.
16-22	Reserved	
23	MMX™ Technology	Processor supports the MMX instruction set. These instructions operate in parallel on multiple data elements (8 bytes, 4 words, or 2 doublewords) packed into quadword registers or memory locations.
24-31	Reserved	

When the input value is 2, the processor returns information about the processor's internal caches and TLBs in the EAX, EBX, ECX, and EDX registers. The encoding of these registers is as follows:

- The least-significant byte in register EAX (register AL) indicates the number of times the CPUID instruction must be executed with an input value of 2 to get a complete description of the processor's caches and TLBs. The Pentium® Pro family of processors will return a 1.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (cleared to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. Table 41-4 shows the encoding of these descriptors.

Table 41-4. Encoding of Cache and TLB Descriptors

Descriptor Value	Cache or TLB Description
00H	Null descriptor
01H	Instruction TLB: 4K-Byte Pages, 4-way set associative, 32 entries
02H	Instruction TLB: 4M-Byte Pages, 4-way set associative, 4 entries
03H	Data TLB: 4K-Byte Pages, 4-way set associative, 64 entries
04H	Data TLB: 4M-Byte Pages, 4-way set associative, 8 entries
06H	Instruction cache: 8K Bytes, 4-way set associative, 32 byte line size
08H	Instruction cache: 16K Bytes, 4-way set associative, 32 byte line size
0AH	Data cache: 8K Bytes, 2-way set associative, 32 byte line size
0CH	Data cache: 16K Bytes, 2-way set associative, 32 byte line size
41H	Unified cache: 128K Bytes, 4-way set associative, 32 byte line size
42H	Unified cache: 256K Bytes, 4-way set associative, 32 byte line size
43H	Unified cache: 512K Bytes, 4-way set associative, 32 byte line size
44H	Unified cache: 1M Byte, 4-way set associative, 32 byte line size

The first member of the Pentium Pro processor family will return the following information about caches and TLBs when the CPUID instruction is executed with an input value of 2:

EAX 03 02 01 01H
EBX 0H
ECX 0H
EDX 06 04 0A 42H

These values are interpreted as follows:

- The least-significant byte (byte 0) of register EAX is set to 01H, indicating that the CPUID instruction needs to be executed only once with an input value of 2 to retrieve complete information about the processor's caches and TLBs.
- The most-significant bit of all four registers (EAX, EBX, ECX, and EDX) is set to 0, indicating that each register contains valid 1-byte descriptors.
- Bytes 1, 2, and 3 of register EAX indicate that the processor contains the following:
  - 01H—A 32-entry instruction TLB (4-way set associative) for mapping 4-KByte pages.
  - 02H—A 4-entry instruction TLB (4-way set associative) for mapping 4-MByte pages.
  - 03H—A 64-entry data TLB (4-way set associative) for mapping 4-KByte pages.
- The descriptors in registers EBX and ECX are valid, but contain null descriptors.
- Bytes 0, 1, 2, and 3 of register EDX indicate that the processor contains the following:
  - 42H—A 256-KByte unified cache (the L2 cache), 4-way set associative, with a 32-byte cache line size.
  - 0AH—An 8-KByte data cache (the L1 data cache), 2-way set associative, with a 32-byte cache line size.
  - 04H—An 8-entry data TLB (4-way set associative) for mapping 4M-byte pages.

— 06H—An 8-KByte instruction cache (the L1 instruction cache), 4-way set associative, with a 32-byte cache line size.

Intel Architecture Compatibility

The CPUID instruction is not supported in early models of the Intel486 processor or in any Intel Architecture processor earlier than the Intel486 processor. The ID flag in the EFLAGS register can be used to determine if this instruction is supported. If a procedure is able to set or clear this flag, the CPUID is supported by the processor running the procedure.

Operation

```
CASE (EAX) OF
    EAX = 0:
        EAX ← highest input value understood by CPUID; (* 2 for Pentium Pro processor *)
        EBX ← Vendor identification string;
        EDX ← Vendor identification string;
        ECX ← Vendor identification string;
    BREAK;
    EAX = 1:
        EAX[3:0] ← Stepping ID;
        EAX[7:4] ← Model;
        EAX[11:8] ← Family;
        EAX[13:12] ← Processor type;
        EAX[31:12] ← Reserved;
        EBX ← Reserved;
        ECX ← Reserved;
        EDX ← Feature flags;  (* See Figure 41-1 *)
    BREAK;
    EAX = 2:
        EAX ← Cache and TLB information;
        EBX ← Cache and TLB information;
        ECX ← Cache and TLB information;
        EDX ← Cache and TLB information;
    BREAK;
    DEFAULT: (* EAX > highest value recognized by CPUID *)
        EAX ← reserved, undefined;
        EBX ← reserved, undefined;
        ECX ← reserved, undefined;
        EDX ← reserved, undefined;
    BREAK;
ESAC;
```

Flags Affected

None.

Exceptions (All Operating Modes)

None.

41.15 CWD/CDQ—Convert Word to Doubleword/Convert Doubleword to Quadword

Opcode	Instruction	Description
99	CWD	DX:AX ← sign-extend of AX
99	CDQ	EDX:EAX ← sign-extend of EAX

Description

Doubles the size of the operand in register AX or EAX (depending on the operand size) by means of sign extension and stores the result in registers DX:AX or EDX:EAX, respectively. The CWD instruction copies the sign (bit 15) of the value in the AX register into every bit position in the DX register (see“Sign Extension”). The CDQ instruction copies the sign (bit 31) of the value in the EAX register into every bit position in the EDX register.

The CWD instruction can be used to produce a doubleword dividend from a word before a word division, and the CDQ instruction can be used to produce a quadword dividend from a doubleword before doubleword division.

The CWD and CDQ mnemonics reference the same opcode. The CWD instruction is intended for use when the operand-size attribute is 16 and the CDQ instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when CWD is used and to 32 when CDQ is used. Others may treat these mnemonics as synonyms (CWD/CDQ) and use the current setting of the operand-size attribute to determine the size of values to be converted, regardless of the mnemonic used.

Operation

```
IF OperandSize = 16 (* CWD instruction *)
    THEN DX ← SignExtend(AX);
    ELSE (* OperandSize = 32, CDQ instruction *)
        EDX ← SignExtend(EAX);
FI;
```

Flags Affected

None.

Exceptions (All Operating Modes)

None.

41.16 CWDE—Convert Word to Doubleword

See entry for CBW/CWDE—Convert Byte to Word/Convert Word to Doubleword.